Realtime Database

Note

The Realtime Database API currently does not support realtime event listeners.

Initializing the Realtime Database component

With the SDK

```
$database = $factory->createDatabase();
```

With Dependency Injection (Symfony Bundle/Laravel/Lumen Package)

```
use Kreait\Firebase\Contract\Database;

class MyService
{
    public function __construct(Database $database)
    {
        $this->database = $database;
    }
}
```

With the Laravel `app()` helper (Laravel/Lumen Package)

```
$database = app('firebase.database');
```

Retrieving data

Every node in your database can be accessed through a Reference:

```
$reference = $database->getReference('path/to/child/location');
```

Note

Creating a reference does not result in a request to your Database. Requests to your Firebase applications are executed with the `getSnapshot()` and `getValue()` methods only.

You can then retrieve a Database Snapshot for the Reference or its value directly:

```
$snapshot = $reference->getSnapshot();

$value = $snapshot->getValue();
// or
$value = $reference->getValue();
```

Database Snapshots

Database Snapshots are immutable copies of the data at a Firebase Database location at the time of a query. The can't be modified and will never change.

```
$snapshot = $reference->getSnapshot();
$value = $snapshot->getValue();

$value = $reference->getValue(); // Shortcut for $reference->getSnapshot()->getValue();
```

Snapshots provide additional methods to work with and analyze the contained value:

- `exists()` returns true if the Snapshot contains any (non-null) data.
- `getChild()` returns another Snapshot for the location at the specified relative path.
- `getKey()` returns the key (last part of the path) of the location of the Snapshot.
- `getReference()` returns the Reference for the location that generated this Snapshot.
- `getValue()` returns the data contained in this Snapshot.
- `hasChild()` returns true if the specified child path has (non-null) data.
- `hasChildren()` returns true if the Snapshot has any child properties, i.e. if the value is an array.
- `numChildren()` returns the number of child properties of this Snapshot, if there are any.

Queries

You can use Queries to filter and order the results returned from the Realtime Database. Queries behave exactly like References. That means you can execute any method on a Query that you can execute on a Reference.

Note

You can combine every filter query with every order query, but not multiple queries of each type. Shallow queries are a special case: they can not be combined with any other query method.

Shallow queries

This is an advanced feature, designed to help you work with large datasets without needing to download everything. Set this to true to limit the depth of the data returned at a location. If the data at the location is a JSON primitive (string, number or boolean), its value will simply be returned.

If the data snapshot at the location is a JSON object, the values for each key will be truncated to true.

Detailed information can be found on the official Firebase documentation page for shallow queries

```
$database->getReference('currencies')
    // order the reference's children by their key in ascending order
    ->shallow()
    ->getSnapshot();
```

A convenience method is available to retrieve the key names of a reference's children:

```
$database->getReference('currencies')->getChildKeys(); // returns an array of key names
```

Ordering data

The official Firebase documentation explains How data is ordered.

Data is always ordered in ascending order.

You can only order by one property at a time - if you try to order by multiple properties, e.g. by child and by value, an exception will be thrown.

By key

```
$database->getReference('currencies')
    // order the reference's children by their key in ascending order
    ->orderByKey()
    ->getSnapshot();
```

By value

Note

In order to order by value, you must define an index, otherwise the Firebase API will refuse the query.

```
{
    "currencies": {
        ".indexOn": ".value"
    }
}
```

```
$database->getReference('currencies')
    // order the reference's children by their value in ascending order
    ->orderByValue()
    ->getSnapshot();
```

By child

Note

In order to order by a child value, you must define an index, otherwise the Firebase API will refuse the query.

```
{
    "people": {
        ".indexOn": "height"
    }
}
```

```
$database->getReference('people')
    // order the reference's children by the values in the field 'height' in ascending order
    ->orderByChild('height')
    ->getSnapshot();
```

Filtering data

To be able to filter results, you must also define an order.

limitToFirst

```php
$database->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // limits the result to the first 10 children (in this case: the 10 shortest persons)
    // values for 'height')
    ->limitToFirst(10)
    ->getSnapshot();
```

limitToLast

```php
$database->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // limits the result to the last 10 children (in this case: the 10 tallest persons)
    ->limitToLast(10)
    ->getSnapshot();
```

startAt

```php
$database->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons taller than or exactly 1.68 (meters)
    ->startAt(1.68)
    ->getSnapshot();
```

startAfter

> **Note**
>
> The `startAfter` query filter has been added to the Firebase JS SDK on 2021-02-11. This PHP SDK implements this in the same way as the other filters, but `startAfter` does not seem to have an effect when used with the Firebase REST API. If you happen to know why or you tried it and it does indeed work, please let me know via the SDK's git repo.

```php
$database->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons taller than 1.68 (meters)
    ->startAfter(1.68)
    ->getSnapshot();
```

endAt

```php
$database->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons shorter than or exactly 1.98 (meters)
    ->endAt(1.98)
    ->getSnapshot();
```

endBefore

> Note
>
> The `endBefore` query filter has been added to the Firebase JS SDK on 2021-02-11. This PHP SDK implements this in the same way as the other filters, but `endBefore` does not seem to have an effect when used with the Firebase REST API. If you happen to know why or you tried it and it does indeed work, please let me know via the SDK's git repo.

```php
$database->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons shorter than 1.98 (meters)
    ->endBefore(1.98)
    ->getSnapshot();
```

equalTo

```php
$database->getReference('people')
    // order the reference's children by the values in the field 'height'
    ->orderByChild('height')
    // returns all persons being exactly 1.98 (meters) tall
    ->equalTo(1.98)
    ->getSnapshot();
```

Saving data

Set/replace values

For basic write operations, you can use set() to save data to a specified reference, replacing any existing data at that path. For example a configuration array for a website might be set as follows:

```php
$database->getReference('config/website')
    ->set([
        'name' => 'My Application',
        'emails' => [
            'support' => 'support@domain.example',
            'sales' => 'sales@domain.example',
        ],
        'website' => 'https://app.domain.example',
    ]);

$database->getReference('config/website/name')->set('New name');
```

Note

Using `set()` overwrites data at the specified location, including any child nodes.

Update specific fields

To simultaneously write to specific children of a node without overwriting other child nodes, use the update() method.

When calling `update()`, you can update lower-level child values by specifying a path for the key. If data is stored in multiple locations to scale better, you can update all instances of that data using data fan-out.

For example, in a blogging app you might want to add a post and simultaneously update it to the recent activity feed and the posting user's activity feed using code like this:

```php
$uid = 'some-user-id';
$postData = [
    'title' => 'My awesome post title',
    'body' => 'This text should be longer',
];

// Create a key for a new post
$newPostKey = $database->getReference('posts')->push()->getKey();

$updates = [
    'posts/'.$newPostKey => $postData,
    'user-posts/'.$uid.'/'.$newPostKey => $postData,
];

$database->getReference() // this is the root reference
    ->update($updates);
```

Writing lists

Use the `push()` method to append data to a list in multiuser applications. The `push()` method generates a unique key every time a new child is added to the specified Firebase reference. By using these auto-generated keys for each new element in the list, several clients

can add children to the same location at the same time without write conflicts. The unique key generated by `push()` is based on a timestamp, so list items are automatically ordered chronologically.

You can use the reference to the new data returned by the `push()` method to get the value of the child's auto-generated key or set data for the child. The `getKey()` method of a `push()` reference contains the auto-generated key.

```
$postData = [...];
$postRef = $database->getReference('posts')->push($postData);

$postKey = $postRef->getKey(); // The key looks like this: -KVquJHezVLf-lSye6Qg
```

Server values

Server values can be written at a location using a placeholder value which is an object with a single `.sv` key. The value for that key is the type of server value you wish to set.

Firebase currently supports only one server value: `timestamp`. You can either set it manually in your write operation, or use a constant from the `Firebase\Database` class.

The following to usages are equivalent:

```
$ref = $database->getReference('posts/my-post')
          ->set('created_at', ['.sv' => 'timestamp']);

$ref = $database->getReference('posts/my-post')
          ->set('created_at', Database::SERVER_TIMESTAMP);
```

Delete data

You can delete a reference, including all data it contains, with the `remove()` method:

```
$database->getReference('posts')->remove();
```

You can also delete by specifying null as the value for another write operation such as `set()` or `update()`.

```
$database->getReference('posts')->set(null);
```

You can also delete in bulk using the function `removeChildren()`:

```
$data->getReference()->removeChildren([
    'posts/post1',
    'user-posts/f55dee9a-dd67-4e78-bd7d-f1b4be157a53/post1',
    'users/f55dee9a-dd67-4e78-bd7d-f1b4be157a53'
]);
```

Database transactions

You can use transaction to update data according to its existing state. For example, if you want to increase an upvote counter, and want to make sure the count accurately reflects multiple, simultaneous upvotes, use a transaction to write the new value to the counter. Instead of two writes that change the counter to the same number, one of the write requests fails and you can then retry the request with the new value.

Replace data inside a transaction

```
use Kreait\Firebase\Database\Transaction;

$counterRef = $database->getReference('counter');

$result = $database->runTransaction(function (Transaction $transaction) use ($counterRef) {

    // You have to snapshot the reference in order to change its value
    $counterSnapshot = $transaction->snapshot($counterRef);

    // Get the existing value from the snapshot
    $counter = $counterSnapshot->getValue() ?: 0;
    $newCounter = ++$counter;

    // If the value hasn't changed in the Realtime Database while we are
    // incrementing it, the transaction will be a success.
    $transaction->set($counterRef, $newCounter);

    return $newCounter;
});
```

Delete data inside a transaction

Likewise, you can wrap the removal of a reference in a transaction as well: you can remove the reference only if it hasn't changed in the meantime.

```
use Kreait\Firebase\Database\Transaction;

$toBeDeleted = $database->getReference('to-be-deleted');

$database->runTransaction(function (Transaction $transaction) use ($toBeDeleted) {

    $transaction->snapshot($toBeDeleted);

    $transaction->remove($toBeDeleted);
});
```

Handling transaction failures

If you haven't snapshotted a reference before trying to change it, the operation will fail with a `\Kreait\Firebase\Exception\Database\ReferenceHasNotBeenSnapshotted` error.

If the reference has changed in the Realtime Database after you started the transaction, the transaction will fail with a `\Kreait\Firebase\Exception\Database\TransactionFailed` error.

```
use Kreait\Firebase\Database\Transaction;
use Kreait\Firebase\Exception\Database\ReferenceHasNotBeenSnapshotted;
use Kreait\Firebase\Exception\Database\TransactionFailed;

$ref = $database->getReference('my-ref');

try {
    $database->runTransaction(function (Transaction $transaction) use ($ref) {

        // $transaction->snapshot($ref);

        $ref->set('value change without a transaction');

        $transaction->set($ref, 'this will fail');
    });

} catch (ReferenceHasNotBeenSnapshotted $e) {

    $referenceInQuestion = $e->getReference();

    echo $e->getReference()->getUri().': '.$e->getMessage();

} catch (TransactionFailed $e) {

    $referenceInQuestion = $e->getReference();
    $failedRequest = $e->getRequest();
    $failureResponse = $e->getResponse();

    echo $e->getReference()->getUri().': '.$e->getMessage();

}
```

Debugging API exceptions

When a request to Firebase fails, the SDK will throw a `\Kreait\Firebase\Exception\ApiException` that includes the sent request and the received response object:

```php
try {
    $database->getReference('forbidden')->getValue();
} catch (ApiException $e) {
    /** @var \Psr\Http\Message\RequestInterface $request */
    $request = $e->getRequest();
    /** @var \Psr\Http\Message\ResponseInterface|null $response */
    $response = $e->getResponse();

    echo $request->getUri().PHP_EOL;
    echo $request->getBody().PHP_EOL;

    if ($response) {
        echo $response->getBody();
    }
}
```

Database rules

Learn more about the usage of Firebase Realtime Database Rules in the official documentation.

```php
use Kreait\Firebase\Database\RuleSet;

// The default rules allow full read and write access to authenticated users of your app
$ruleSet = RuleSet::default();

// This level of access means anyone can read or write to your database. You should
// configure more secure rules before launching your app.
$ruleSet = RuleSet::public();

// Private rules disable read and write access to your database by users.
// With these rules, you can only access the database through the
// Firebase console and the Admin SDKs.
$ruleSet = RuleSet::private();

// You can define custom rules
$ruleSet = RuleSet::fromArray(['rules' => [
    '.read' => true,
    '.write' => false,
    'users' => [
        '$uid' => [
            '.read' => '$uid === auth.uid',
            '.write' => '$uid === auth.uid',
        ]
    ]
]]);

$database->updateRules($ruleSet);

$freshRuleSet = $database->getRuleSet(); // Returns a new RuleSet instance
$actualRules = $ruleSet->getRules(); // returns an array
```

Authenticate with limited privileges

As a best practice, a service should have access to only the resources it needs. To get more fine-grained control over the resources a Firebase app instance can access, use a unique identifier in your Security Rules to represent your service. Then set up appropriate rules which grant your service access to the resources it needs. For example:

```
{
  "rules": {
    "public_resource": {
      ".read": true,
      ".write": true
    },
    "some_resource": {
      ".read": "auth.uid === 'my-service-worker'",
      ".write": false
    },
    "another_resource": {
      ".read": "auth.uid === 'my-service-worker'",
      ".write": "auth.uid === 'my-service-worker'"
    }
  }
}
```

Then, when instantiating the database component of the SDK, use the `withDatabaseAuthVariableOverride()` method to override the auth object used by your database rules. In this custom auth object, set the `uid` field to the identifier you used to represent your service in your Security Rules.

```
use Kreait\Firebase\Factory;

$factory = (new Factory)
    ->withServiceAccount('/path/to/firebase_credentials.json')
    ->withDatabaseUri('https://my-project-default-rtdb.firebaseio.com');

$database = $factory
    ->withDatabaseAuthVariableOverride('my-service-worker')
    ->createDatabase();

// $database now only has access as defined in the Security Rules
```

In some cases, you may want to downscope the Admin SDKs to act as an unauthenticated client. You can do this by providing a value of `null` for the database auth variable override.

```
$database = $factory
    ->withDatabaseAuthVariableOverride(null)
    ->createDatabase();

// $database now only has access to public resources
```